Statistical Science
2010, Vol. 25, No. 3, 311–324
DOI: 10.1214/10-STS336

Graphics Processing Units and High-Dimensional Optimization

Hua Zhou, Kenneth Lange and Marc A. Suchard

Abstract. This article discusses the potential of graphics processing units (GPUs) in high-dimensional optimization problems. A single GPU card with hundreds of arithmetic cores can be inserted in a personal computer and dramatically accelerates many statistical algorithms. To exploit these devices fully, optimization algorithms should reduce to multiple parallel tasks, each accessing a limited amount of data. These criteria favor EM and MM algorithms that separate parameters and data. To a lesser extent block relaxation and coordinate descent and ascent also qualify. We demonstrate the utility of GPUs in nonnegative matrix factorization, PET image reconstruction, and multidimensional scaling. Speedups of 100-fold can easily be attained. Over the next decade, GPUs will fundamentally alter the landscape of computational statistics. It is time for more statisticians to get on-board.

Key words and phrases: Block relaxation, EM and MM algorithms, multidimensional scaling, nonnegative matrix factorization, parallel computing, PET scanning.

1. INTRODUCTION

Statisticians, like all scientists, are acutely aware that the clock speeds on their desktops and laptops have stalled. Does this mean that statistical computing has hit a wall? The answer fortunately is no, but the hardware advances that we routinely expect have taken an interesting detour. Most computers now sold have two to eight processing cores. Think of these as separate CPUs on the same chip. Naive programmers rely on sequential algorithms and often fail to take advantage of more than a single core. Sophisticated programmers eagerly exploit parallel programming. However, multicore CPUs do not represent the only road to success of statistical computing.

Hua Zhou is Assistant Professor, Department of Statistics, North Carolina State University, Raleigh, North Carolina 27695-8203, USA (e-mail: huazhou@ncsu.edu). Kenneth Lange is Professor, Departments of Biomathematics, Human Genetics, and Statistics, UCLA, 5357A Gonda Building, Los Angeles, California 90095-1766, USA (e-mail: klange@ucla.edu). Marc A. Suchard is Professor, Departments of Biomathematics, Biostatistics, and Human Genetics, UCLA, 6558 Gonda Building, Los Angeles, California 90095-1766, USA (e-mail: msuchard@ucla.edu).

Graphics processing units (GPUs) have caught the scientific community by surprise. These devices are designed for graphics rendering in computer animation and games. Propelled by these nonscientific markets, the old technology of numerical (array) coprocessors has advanced rapidly. Highly parallel GPUs are now making computational inroads against traditional CPUs in image processing, protein folding, stock options pricing, robotics, oil exploration, data mining and many other areas [28]. We are starting to see orders of magnitude improvement on some hard computational problems. Three companies, Intel, NVIDIA and AMD/ATI, dominate the market. Intel is struggling to keep up with its more nimble competitors.

Modern GPUs support more vector and matrix operations, stream data faster, and possess more local memory per core than their predecessors. They are also readily available as commodity items that can be inserted as video cards on modern PCs. GPUs have been criticized for their hostile programming environment and lack of double precision arithmetic and error correction, but these faults are being rectified. The CUDA programming environment [27] for NVIDIA chips is now easing some of the programming chores. We could say more about near-term improvements, but most pronouncements would be obsolete within months.

Oddly, statisticians have been slow to embrace the new technology. Silberstein et al. [31] first demonstrated the potential for GPUs in fitting simple Bayesian networks. Recently Suchard and Rambaut [33] have seen greater than 100-fold speedups in MCMC simulations in molecular phylogeny. Lee et al. [18] and Tibbits, Haran and Liechty [36] are following suit with Bayesian model fitting via particle filtering and slice sampling. Finally, work is underway to port common data mining techniques such as hierarchical clustering and multifactor dimensionality reduction onto GPUs [32]. These efforts constitute the first wave of an eventual flood of statistical and data mining applications. The porting of GPU tools into the R environment will undoubtedly accelerate the trend [3].

Not all problems in computational statistics can benefit from GPUs. Sequential algorithms are resistant unless they can be broken into parallel pieces. For example, least squares and singular value decomposition—two tasks frequently performed in statistics—may not benefit greatly from GPUs unless problems are extremely large scale or many small problems need to be solved simultaneously. Even parallel algorithms can be problematic if the entire range of data must be accessed by each GPU. A case in point is the alternating least squares strategy for the nonnegative matrix factorization problem featured in Section 3.1. Because they have limited memory, GPUs are designed to operate on short streams of data. The greatest speedups occur when all of the GPUs on a card perform the same arithmetic operation simultaneously. Effective applications of GPUs in optimization involve both separation of data and separation of parameters.

In the current article, we illustrate how GPUs can work hand in glove with the MM algorithm, a generalization of the EM algorithm. In many optimization problems, the MM algorithm explicitly separates parameters by replacing the objective function by a sum of surrogate functions, each of which involves a single parameter. Optimization of the one-dimensional surrogates can be accomplished by assigning each subproblem to a different core. Provided the different cores each access just a slice of the data, the parallel subproblems execute quickly. By construction the new point in parameter space improves the value of the objective function. In other words, MM algorithms are iterative ascent or descent algorithms. If they are well designed, then they separate parameters in high-dimensional problems. This is where GPUs enter. They offer most of the benefits of distributed computer clusters at a fraction of the cost. For this reason alone, computational statisticians need to pay attention to GPUs.

Before formally defining the MM algorithm, it may help the reader to walk through a simple numerical example stripped of statistical content. Consider the Rosenbrock test function

$$\begin{aligned} f(\mathbf{x}) &= 100(x_1^2 - x_2)^2 + (x_1 - 1)^2 \\ &= 100(x_1^4 + x_2^2 - 2x_1^2x_2) + (x_1^2 - 2x_1 + 1), \end{aligned} \tag{1.1}$$

familiar from the minimization literature. As we iterate toward the minimum at $\mathbf{x} = \mathbf{1} = (1, 1)$, we construct a surrogate function that separates parameters. This is done by exploiting the obvious majorization

$$\begin{aligned} -2x_1^2x_2 \le{}& x_1^4 + x_2^2 + (x_{n1}^2 + x_{n2})^2 \\ &- 2(x_{n1}^2 + x_{n2})(x_1^2 + x_2), \end{aligned}$$

where equality holds when $\mathbf{x}$ and the current iterate $\mathbf{x}_n$ coincide. It follows that $f(\mathbf{x})$ itself is majorized by the sum of the two surrogates

$$\begin{aligned} &g_1(x_1|\mathbf{x}_n) \\ &\quad = 200x_1^4 - [200(x_{n1}^2 + x_{n2}) - 1]x_1^2 - 2x_1 + 1, \\ &g_2(x_2|\mathbf{x}_n) \\ &\quad = 200x_2^2 - 200(x_{n1}^2 + x_{n2})x_2 + (x_{n1}^2 + x_{n2})^2. \end{aligned}$$

The left panel of Figure 1 depicts the Rosenbrock function and its majorization $g_1(x_1|\mathbf{x}_n) + g_2(x_2|\mathbf{x}_n)$ at the point $-\mathbf{1}$.

According to the MM recipe, at each iteration one must minimize the quartic polynomial $g_1(x_1|\mathbf{x}_n)$ and the quadratic polynomial $g_2(x_2|\mathbf{x}_n)$. The quartic possesses either a single global minimum or two local minima separated by a local maximum. These minima are the roots of the cubic function $g_1'(x_1|\mathbf{x}_n)$ and can be explicitly computed. We update x_1 by the root corresponding to the global minimum and x_2 via $x_{n+1,2} = \frac{1}{2}(x_{n1}^2 + x_{n2})$. The right panel of Figure 1 displays the iterates starting from $\mathbf{x}_0 = -\mathbf{1}$. These immediately jump into the Rosenbrock valley and then slowly descend to $\mathbf{1}$.

Separation of parameters in this example makes it easy to decrease the objective function. This almost trivial advantage is amplified when we optimize functions depending on tens of thousands to millions of parameters. In these settings, Newton's method and variants such as Fisher's scoring are fatally handicapped by the need to store, compute, and invert huge Hessian or information matrices. On the negative side of the balance sheet, MM algorithms are often slow to converge. This disadvantage is usually outweighed by the speed of their updates even in sequential mode. If one can

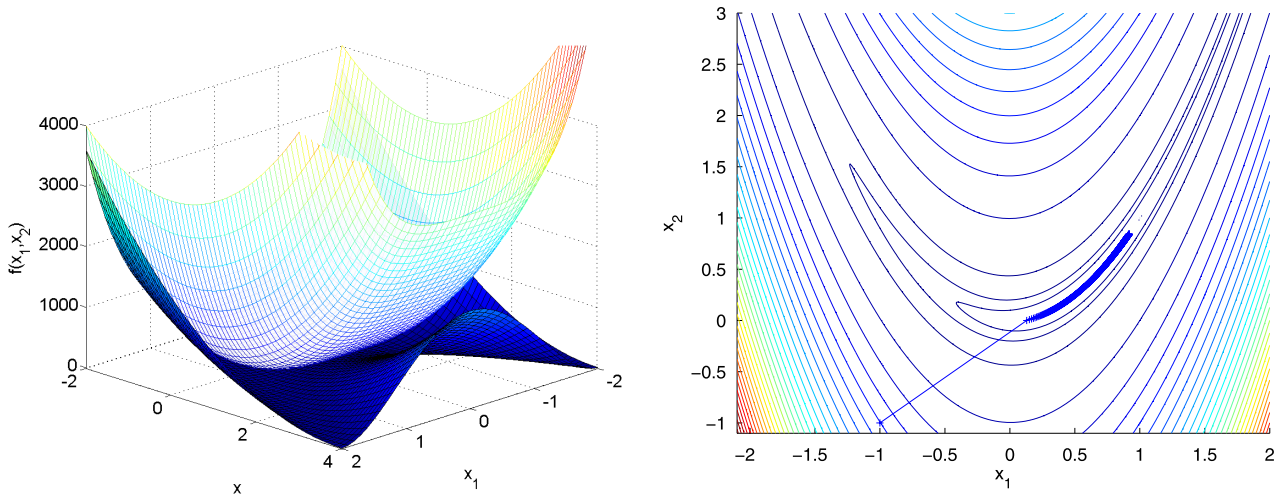

FIG. 1. Left: *The Rosenbrock (banana) function (the lower surface) and a majorization function at point* $(-1, -1)$ *(the upper surface).* Right: *MM iterates.*

harness the power of parallel processing GPUs, then MM algorithms become the method of choice for many high-dimensional problems.

We conclude this introduction by sketching a roadmap to the rest of the article. Section 2 reviews the MM algorithm. Section 3 discusses three high-dimensional MM examples. Although the algorithm in each case is known, we present brief derivations to illustrate how simple inequalities drive separation of parameters. We then implement each algorithm on a realistic problem and compare running times in sequential and parallel modes. We purposefully omit extensive programming syntax since many tutorials already exist for this purpose, and material of this sort is bound to be ephemeral. The two tutorials [34, 35] are a good place to start for statisticians. Section 4 concludes with a brief discussion of other statistical applications of GPUs and other methods of accelerating optimization algorithms.

2. MM ALGORITHMS

The MM algorithm, like the EM algorithm, is a principle for creating optimization algorithms. In minimization the acronym MM stands for majorization–minimization; in maximization it stands for minorization–maximization. Both versions are convenient in statistics. For the moment we will concentrate on maximization.

Let $f(\boldsymbol{\theta})$ be the objective function whose maximum we seek. Its argument $\boldsymbol{\theta}$ can be high-dimensional and vary over a constrained subset Θ of Euclidean space. An MM algorithm involves minorizing $f(\boldsymbol{\theta})$ by a surrogate function $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ anchored at the current iterate $\boldsymbol{\theta}_n$ of the search. The subscript n indicates iteration number throughout this article. If $\boldsymbol{\theta}_{n+1}$ denotes the maximum of $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ with respect to its left argument, then the MM principle declares that $\boldsymbol{\theta}_{n+1}$ increases $f(\boldsymbol{\theta})$ as well. Thus, MM algorithms revolve around a basic ascent property.

Minorization is defined by the two properties

$$f(\boldsymbol{\theta}_n) = g(\boldsymbol{\theta}_n|\boldsymbol{\theta}_n), \tag{2.1}$$

$$f(\boldsymbol{\theta}) \geq g(\boldsymbol{\theta}|\boldsymbol{\theta}_n), \quad \boldsymbol{\theta} \neq \boldsymbol{\theta}_n. \tag{2.2}$$

In other words, the surface $\boldsymbol{\theta} \mapsto g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ lies below the surface $\boldsymbol{\theta} \mapsto f(\boldsymbol{\theta})$ and is tangent to it at the point $\boldsymbol{\theta} = \boldsymbol{\theta}_n$. Construction of the minorizing function $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ constitutes the first M of the MM algorithm. In our examples $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ is chosen to separate parameters.

In the second M of the MM algorithm, one maximizes the surrogate $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ rather than $f(\boldsymbol{\theta})$ directly. It is straightforward to show that the maximum point $\boldsymbol{\theta}_{n+1}$ satisfies the ascent property $f(\boldsymbol{\theta}_{n+1}) \geq f(\boldsymbol{\theta}_n)$. The proof

$$f(\boldsymbol{\theta}_{n+1}) \geq g(\boldsymbol{\theta}_{n+1}|\boldsymbol{\theta}_n) \geq g(\boldsymbol{\theta}_n|\boldsymbol{\theta}_n) = f(\boldsymbol{\theta}_n)$$

reflects definitions (2.1) and (2.2) and the choice of $\boldsymbol{\theta}_{n+1}$. The ascent property is the source of the MM algorithm's numerical stability and remains valid if we merely increase $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ rather than maximize it. In many problems MM updates are delightfully simple to code, intuitively compelling, and automatically consistent with parameter constraints. In minimization we seek a majorizing function $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ lying above the surface $\boldsymbol{\theta} \mapsto f(\boldsymbol{\theta})$ and tangent to it at the point $\boldsymbol{\theta} = \boldsymbol{\theta}_n$. Minimizing $g(\boldsymbol{\theta}|\boldsymbol{\theta}_n)$ drives $f(\boldsymbol{\theta})$ downhill.

The celebrated Expectation–Maximization (EM) algorithm [8, 22] is a special case of the MM algorithm. The Q-function produced in the E step of the EM algorithm constitutes a minorizing function of the log-likelihood. Thus, both EM and MM share the same advantages: simplicity, stability, graceful adaptation to constraints, and the tendency to avoid large matrix inversion. The more general MM perspective frees algorithm derivation from the missing data straitjacket and invites wider applications. For example, our multidimensional scaling (MDS) and nonnegative matrix factorization (NNFM) examples involve no likelihood functions. Wu and Lange [40] briefly summarized the history of the MM algorithm and its relationship to the EM algorithm.

The convergence properties of MM algorithms are well known [16]. In particular, five properties of the objective function $f(\boldsymbol{\theta})$ and the MM algorithm map $\boldsymbol{\theta} \mapsto M(\boldsymbol{\theta})$ guarantee convergence to a stationary point of $f(\boldsymbol{\theta})$: (a) $f(\boldsymbol{\theta})$ is coercive on its open domain; (b) $f(\boldsymbol{\theta})$ has only isolated stationary points; (c) $M(\boldsymbol{\theta})$ is continuous; (d) $\boldsymbol{\theta}^*$ is a fixed point of $M(\boldsymbol{\theta})$ if and only if $\boldsymbol{\theta}^*$ is a stationary point of $f(\boldsymbol{\theta})$; and (e) $f[M(\boldsymbol{\theta}^*)] \geq f(\boldsymbol{\theta}^*)$, with equality if and only if $\boldsymbol{\theta}^*$ is a fixed point of $M(\boldsymbol{\theta})$. These conditions are easy to verify in many applications. The local rate of convergence of an MM algorithm is intimately tied to how well the surrogate function $g(\boldsymbol{\theta}|\boldsymbol{\theta}^*)$ approximates the objective function $f(\boldsymbol{\theta})$ near the optimal point $\boldsymbol{\theta}^*$.

3. NUMERICAL EXAMPLES

In this section, we compare the performances of the CPU and GPU implementations of three classical MM algorithms coded in C++: (a) nonnegative matrix factorization (NNMF), (b) positron emission tomography (PET), and (c) multidimensional scaling (MDS). In each case we briefly derive the algorithm from the MM perspective. For the CPU version, we iterate until the relative change

$$\frac{|f(\boldsymbol{\theta}_n) - f(\boldsymbol{\theta}_{n-1})|}{|f(\boldsymbol{\theta}_{n-1})| + 1}$$

of the objective function $f(\boldsymbol{\theta})$ between successive iterations falls below a pre-set threshold ε or the number of iterations reaches a pre-set number $n_{\max}$, whichever comes first. In these examples, we take $\varepsilon = 10^{-9}$ and $n_{\max} = 100{,}000$. For ease of comparison, we iterate the GPU version for the same number of steps as the CPU version. Overall, we see anywhere from a 22-fold to a 112-fold decrease in total run time on our hardware.

TABLE 1
Configuration of the desktop system

	CPU	GPU
Model	Intel Core 2 Extreme X9440	NVIDIA GeForce GTX 280
# Cores	4	240
Clock	3.2 G	1.3 G
Memory	16 G	1 G

The source code is freely available from the first author.

Table 1 shows how our desktop system is configured. Although the CPU is a high-end processor with four cores, we use just one of these for ease of comparison. In practice, it takes considerable effort to load balance the various algorithms across multiple CPU cores. With 240 GPU cores, the GTX 280 GPU card delivers a peak performance of about 933 GFlops in single precision. This card is already obsolete. Newer cards possess twice as many cores and better double-precision capability. Up to four cards can fit inside a single desktop computer. It is relatively straightforward to program multiple GPUs. Because previous-generation GPU hardware is largely limited to single precision, this is a worry in scientific computing. To assess the extent of roundoff error, we display the converged values of the objective functions to ten significant digits. Only rarely is the GPU value far off the CPU mark. The extra effort in programming the GPU version is relatively light. Exploiting the standard CUDA library [27], it takes 77, 176 and 163 extra lines of GPU code to implement the NNMF, PET, and MDS examples, respectively. Finally, for the PET and MDS examples, we also list run times of a CPU implementation with a quasi-Newton acceleration [42]. This generic acceleration significantly reduces the number of MM iterations until convergence.

3.1 Nonnegative Matrix Factorizations

Nonnegative matrix factorization (NNMF) is an alternative to principal component analysis useful in modeling, compressing, and interpreting nonnegative data such as observational counts and images. The articles [2, 19, 20] discuss in detail algorithm development and statistical applications of NNMF. The basic problem is to approximate a data matrix $\mathbf{X}$ with nonnegative entries x_{ij} by a product $\mathbf{VW}$ of two low-rank matrices $\mathbf{V}$ and $\mathbf{W}$ with nonnegative entries v_{ik} and w_{kj}. Here $\mathbf{X}$, $\mathbf{V}$ and $\mathbf{W}$ are $p \times q$, $p \times r$ and $r \times q$, respectively,

with r much smaller than $\min\{p, q\}$. One version of NNMF minimizes the objective function

$$f(\mathbf{V}, \mathbf{W}) = \|\mathbf{X} - \mathbf{V}\mathbf{W}\|_{\mathrm{F}}^2 = \sum_i \sum_j \left(x_{ij} - \sum_k v_{ik} w_{kj}\right)^2, \tag{3.1}$$

where $\|\cdot\|_{\mathrm{F}}$ denotes the Frobenius norm. To get an idea of the scale of NNFM imaging problems, p (number of images) can range 10^1 to 10^4, q (number of pixels per image) can surpass 10^2 to 10^4, and one seeks a rank-r approximation of about 50. Notably, part of the winning solution of the Netflix challenge relies on variations of NNMF [13]. For the Netflix data matrix, $p = 480{,}000$ (raters), $q = 18{,}000$ (movies), and r ranged from 20 to 100.

Exploiting the convexity of the function $x \mapsto (x_{ij} - x)^2$, one can derive the inequality

$$\left(x_{ij} - \sum_k v_{ik} w_{kj}\right)^2 \le \sum_k \frac{a_{nikj}}{b_{nij}} \left(x_{ij} - \frac{b_{nij}}{a_{nikj}} v_{ik} w_{kj}\right)^2,$$

where $a_{nikj} = v_{nik} w_{nkj}$ and $b_{nij} = \sum_k a_{nikj}$. This leads to the surrogate function

$$g(\mathbf{V}, \mathbf{W} | \mathbf{V}_n, \mathbf{W}_n) = \sum_i \sum_j \sum_k \frac{a_{nikj}}{b_{nij}} \left(x_{ij} - \frac{b_{nij}}{a_{nikj}} v_{ik} w_{kj}\right)^2 \tag{3.2}$$

majorizing the objective function $f(\mathbf{V}, \mathbf{W}) = \|\mathbf{X} - \mathbf{V}\mathbf{W}\|_{\mathrm{F}}^2$. Although the majorization (3.2) does not achieve a complete separation of parameters, it does if we fix $\mathbf{V}$ and update $\mathbf{W}$ or vice versa. This strategy is called block relaxation [7].

If we elect to minimize $g(\mathbf{V}, \mathbf{W} | \mathbf{V}_n, \mathbf{W}_n)$ holding $\mathbf{W}$ fixed at $\mathbf{W}_n$, then the stationarity condition for $\mathbf{V}$ reads

$$\frac{\partial}{\partial v_{ik}} g(\mathbf{V}, \mathbf{W}_n | \mathbf{V}_n, \mathbf{W}_n) = -2 \sum_j \left(x_{ij} - \frac{b_{nij}}{a_{nikj}} v_{ik} w_{nkj}\right) w_{nkj} = 0.$$

Its solution furnishes the simple multiplicative update

$$v_{n+1,ik} = v_{nik} \frac{\sum_j x_{ij} w_{nkj}}{\sum_j b_{nij} w_{nkj}}. \tag{3.3}$$

Likewise the stationary condition

$$\frac{\partial}{\partial w_{kj}} g(\mathbf{V}_{n+1}, \mathbf{W} | \mathbf{V}_{n+1}, \mathbf{W}_n) = 0$$

gives the multiplicative update

$$w_{n+1,kj} = w_{nkj} \frac{\sum_i x_{ij} v_{n+1,ik}}{\sum_i c_{nij} v_{n+1,ik}}, \tag{3.4}$$

where $c_{nij} = \sum_k v_{n+1,ik} w_{nkj}$. Close inspection of the multiplicative updates (3.3) and (3.4) shows that their numerators depend on the matrix products $\mathbf{X}\mathbf{W}_n^t$ and $\mathbf{V}_{n+1}^t \mathbf{X}$ and their denominators depend on the matrix products $\mathbf{V}_n \mathbf{W}_n \mathbf{W}_n^t$ and $\mathbf{V}_{n+1}^t \mathbf{V}_{n+1} \mathbf{W}_n$. Large matrix multiplications are very fast on GPUs because CUDA implements in parallel the BLAS (basic linear algebra subprograms) library widely applied in numerical analysis [26]. Once the relevant matrix products are available, each elementwise update of v_{ik} or w_{kj} involves just a single multiplication and division. These scalar operations are performed in parallel through handwritten GPU code. Algorithm 1 summarizes the steps in performing NNMF, and Listing 1 illustrates the scalar operation kernel.

We now compare CPU and GPU versions of the multiplicative NNMF algorithm on a training set of face images. Database #1 from the MIT Center for Biological and Computational Learning (CBCL) [25] reduces to a matrix $\mathbf{X}$ containing $p = 2429$ gray-scale face images with $q = 19 \times 19 = 361$ pixels per face. Each image (row) is scaled to have mean and standard deviation 0.25. Figure 2 shows the recovery of the first face in the database using a rank 49 decomposition. The 49 basis images (rows of $\mathbf{W}$) represent different aspects of a face. The rows of $\mathbf{V}$ contain the coefficients of these parts estimated for the various faces. Some of these facial features are immediately obvious in the reconstruction. Table 2 compares the run times of Algorithm 1 implemented on our CPU and GPU, respectively. We observe a 22- to 112-fold speed-up in the GPU implementation. Run times for

Algorithm 1 (NNMF). Given $\mathbf{X} \in \mathbb{R}_+^{p \times q}$, find $\mathbf{V} \in \mathbb{R}_+^{p \times r}$ and $\mathbf{W} \in \mathbb{R}_+^{r \times q}$ minimizing $\|\mathbf{X} - \mathbf{V}\mathbf{W}\|_{\mathrm{F}}^2$.

Initialize: Draw v_{0ik} and w_{0kj} uniform on (0,1) for all $1 \le i \le p$, $1 \le k \le r$, $1 \le j \le q$
repeat
  Compute $\mathbf{X}\mathbf{W}_n^t$ and $\mathbf{V}_n \mathbf{W}_n \mathbf{W}_n^t$
  $v_{n+1,ik} \leftarrow v_{nik} \cdot \{\mathbf{X}\mathbf{W}_n^t\}_{ik} / \{\mathbf{V}_n \mathbf{W}_n \mathbf{W}_n^t\}_{ik}$ for all $1 \le i \le p$, $1 \le k \le r$
  Compute $\mathbf{V}_{n+1}^t \mathbf{X}$ and $\mathbf{V}_{n+1}^t \mathbf{V}_{n+1} \mathbf{W}_n$
  $w_{n+1,kj} \leftarrow w_{nkj} \cdot \{\mathbf{V}_{n+1}^t \mathbf{X}\}_{kj} / \{\mathbf{V}_{n+1}^t \mathbf{V}_{n+1} \mathbf{W}_n\}_{kj}$ for all $1 \le k \le r$, $1 \le j \le q$
until convergence occurs

```
__global__ void update_V_kernel(
                        Real* V, Real* XWt, Real* VWWt,
                        int p, int r, int stride
        ){

        // Determine indices for this thread,
           column-major storage
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int k = blockIdx.y * blockDim.y + threadIdx.y;
        int ik = i + k * stride;

        // Perform scalar operation on each matrix entry
        if ( i < p && k < r ) {
                V[ik] = V[ik] * XWt[ik] / VWWt[ik];
    }
}
```

LISTING 1. *Scalar operation GPU kernel for NNMF.*

the GPU version depend primarily on the number of iterations to convergence and very little on the rank r of the approximation. Run times of the CPU version scale linearly in both the number of iterations and r.

It is worth stressing a few points. First, the objective function (3.1) is convex in $\mathbf{V}$ for $\mathbf{W}$ fixed, and vice versa, but not jointly convex. Thus, even though the MM algorithm enjoys the descent property, it is not guaranteed to find the global minimum [2]. There are two good alternatives to the multiplicative algorithm. First, pure block relaxation can be conducted by alternating least squares (ALS). In updating $\mathbf{V}$ with $\mathbf{W}$ fixed, ALS omits majorization and solves the p separated nonnegative least squares problems

$$\min_{\mathbf{V}(i,:)} \|\mathbf{X}(i,:) - \mathbf{V}(i,:)\mathbf{W}\|_2^2$$
$$\text{subject to} \quad \mathbf{V}(i,:) \geq 0,$$

where $\mathbf{V}(i,:)$ and $\mathbf{X}(i,:)$ denote the ith row of the corresponding matrices. Similarly, in updating $\mathbf{W}$ with $\mathbf{V}$ fixed, ALS solves q separated nonnegative least squares problems. Separation naturally suggests parallel implementations, but parallelization on a GPU can hit a snag because each nonnegative least squares subproblem needs to operate on the whole $\mathbf{W}$ matrix simultaneously. Another possibility is to change the objective function to

$$L(\mathbf{V}, \mathbf{W}) = \sum_i \sum_j \left[x_{ij} \ln\left(\sum_k v_{ik} w_{kj} \right) - \sum_k v_{ik} w_{kj} \right]$$

according to a Poisson model for the counts x_{ij} [19]. This works even when some entries x_{ij} fail to be integers, but the Poisson log-likelihood interpretation is lost. A pure MM algorithm for maximizing $L(\mathbf{V}, \mathbf{W})$ is

$$v_{n+1,ik} = v_{nik} \sqrt{\frac{\sum_j x_{ij} w_{nkj} / b_{nij}}{\sum_j w_{nkj}}},$$
$$w_{n+1,ij} = w_{nkj} \sqrt{\frac{\sum_i x_{ij} v_{nik} / b_{nij}}{\sum_i v_{nik}}}.$$

Derivation of these variants of Lee and Seung's [19] Poisson updates is left to the reader.

3.2 Positron Emission Tomography

The field of computed tomography has exploited EM algorithms for many years. In positron emission tomography (PET), the reconstruction problem consists of estimating the Poisson emission intensities $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_p)$ of p pixels arranged in a two-dimensional grid surrounded by an array of photon detectors. The observed data are coincidence counts $(y_1, \ldots, y_d)$ along d lines of flight connecting pairs of photon detectors. The log-likelihood under the PET model is

$$L(\boldsymbol{\lambda}) = \sum_i \left[y_i \ln\left(\sum_j e_{ij} \lambda_j \right) - \sum_j e_{ij} \lambda_j \right],$$

where the e_{ij} are constants derived from the geometry of the grid and the detectors. Without loss of generality,

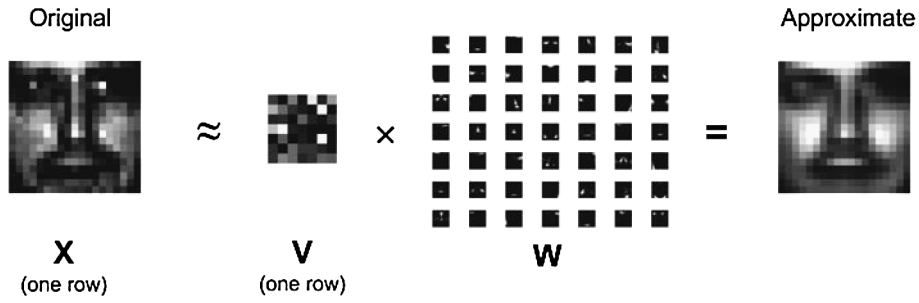

FIG. 2. *Approximation of a face image by rank-49 NNMF: coefficients × basis images = approximate image.*

TABLE 2
Run-time (in seconds) comparisons for NNMF on the MIT CBCL face image data

Rank r	Iters	CPU		GPU		Speedup
		Time	Function	Time	Function	
10	25,459	1203	106.2653503	55	106.2653504	22
20	87,801	7564	89.56601262	163	89.56601287	46
30	55,783	7013	78.42143486	103	78.42143507	68
40	47,775	7880	70.05415929	119	70.05415950	66
50	53,523	11,108	63.51429261	121	63.51429219	92
60	77,321	19,407	58.24854375	174	58.24854336	112

Notes: The dataset contains $p = 2429$ faces with $q = 19 \times 19 = 361$ pixels per face. The columns labeled Function refer to the converged value of the objective function.

one can assume $\sum_i e_{ij} = 1$ for each j. It is straightforward to derive the traditional EM algorithm [14, 39] from the MM perspective using the concavity of the function $\ln s$. Indeed, application of Jensen's inequality produces the minorization

$$\begin{aligned} L(\boldsymbol{\lambda}) &\geq \sum_i y_i \sum_j w_{nij} \ln\left(\frac{e_{ij}\lambda_j}{w_{nij}}\right) - \sum_i \sum_j e_{ij}\lambda_j \\ &= Q(\boldsymbol{\lambda}|\boldsymbol{\lambda}_n), \end{aligned}$$

where $w_{nij} = e_{ij}\lambda_{nj}/(\sum_k e_{ik}\lambda_{nk})$. This maneuver again separates parameters. The stationarity conditions for the surrogate $Q(\boldsymbol{\lambda}|\boldsymbol{\lambda}_n)$ supply the parallel updates

$$\lambda_{n+1,j} = \frac{\sum_i y_i w_{nij}}{\sum_i e_{ij}}. \tag{3.5}$$

The convergence of the PET algorithm (3.5) is frustratingly slow, even under systematic acceleration [30, 42]. Furthermore, the reconstructed images are of poor quality with a grainy appearance. The early remedy of premature halting of the algorithm cuts computational cost but is entirely ad hoc, and the final image depends on initial conditions. A better option is to add a roughness penalty to the log-likelihood. This device not only produces better images but also accelerates convergence. Thus, we maximize the penalized log-likelihood

$$f(\boldsymbol{\lambda}) = L(\boldsymbol{\lambda}) - \frac{\mu}{2} \sum_{\{j,k\}\in\mathcal{N}} (\lambda_j - \lambda_k)^2, \tag{3.6}$$

where μ is the roughness penalty constant, and $\mathcal{N}$ is the neighborhood system that pairs spatially adjacent pixels. An absolute value penalty is less likely to deter the formation of edges than a square penalty, but it is easier to deal with a square penalty analytically, and we adopt it for the sake of simplicity. In practice, visual inspection of the recovered images guides the selection of the roughness penalty constant μ.

To maximize $f(\boldsymbol{\lambda})$ by an MM algorithm, we must minorize the penalty in a manner consistent with the separation of parameters. In view of the evenness and convexity of the function s^2, we have

$$\begin{aligned} (\lambda_j - \lambda_k)^2 &\leq \tfrac{1}{2}(2\lambda_j - \lambda_{nj} - \lambda_{nk})^2 \\ &\quad + \tfrac{1}{2}(2\lambda_k - \lambda_{nj} - \lambda_{nk})^2. \end{aligned}$$

Equality holds if $\lambda_j + \lambda_k = \lambda_{nj} + \lambda_{nk}$, which is true when $\boldsymbol{\lambda} = \boldsymbol{\lambda}_n$. Combining our two minorizations furnishes the surrogate function

$$\begin{aligned} &g(\boldsymbol{\lambda}|\boldsymbol{\lambda}_n) \\ &\quad = Q(\boldsymbol{\lambda}|\boldsymbol{\lambda}_n) - \frac{\mu}{4} \sum_{\{j,k\}\in\mathcal{N}} [(2\lambda_j - \lambda_{nj} - \lambda_{nk})^2 \\ &\qquad\qquad + (2\lambda_k - \lambda_{nj} - \lambda_{nk})^2]. \end{aligned}$$

To maximize $g(\boldsymbol{\lambda}|\boldsymbol{\lambda}_n)$, we define $\mathcal{N}_j = \{k : \{j,k\} \in \mathcal{N}\}$ and set the partial derivative

$$\begin{aligned} \frac{\partial}{\partial\lambda_j} g(\boldsymbol{\lambda}|\boldsymbol{\lambda}_n) &= \sum_i \left[\frac{y_i w_{nij}}{\lambda_j} - e_{ij}\right] \\ &\quad - \mu \sum_{k:\in\mathcal{N}_j} (2\lambda_j - \lambda_{nj} - \lambda_{nk}) \end{aligned} \tag{3.7}$$

equal to 0 and solve for $\lambda_{n+1,j}$. Multiplying (3.7) by λ_j produces a quadratic with roots of opposite signs. We take the positive root

$$\lambda_{n+1,j} = \frac{-b_{nj} - \sqrt{b_{nj}^2 - 4a_j c_{nj}}}{2a_j},$$

where

$$a_j = -2\mu \sum_{k \in \mathcal{N}_j} 1,$$

$$b_{nj} = \sum_{k \in \mathcal{N}_j} (\lambda_{nj} + \lambda_{nk}) - 1,$$

$$c_{nj} = \sum_i y_i w_{nij}.$$

Algorithm 2 summarizes the complete MM scheme. Obviously, complete parameter separation is crucial. The quantities a_j can be computed once and stored. The quantities b_{nj} and c_{nj} are computed for each j in parallel. To improve GPU performance in computing the sums over i, we exploit the widely available parallel sum-reduction techniques [31]. Given these results, a specialized but simple GPU code computes the updates $\lambda_{n+1,j}$ for each j in parallel.

Table 3 compares the run times of the CPU and GPU implementations for a simulated PET image [30]. The image as depicted in the top of Figure 3 has $p = 64 \times 64 = 4096$ pixels and is interrogated by $d = 2016$ detectors. Overall we see a 43- to 53-fold reduction in run times with the GPU implementation. Figure 3 displays the true image and the estimated images under penalties of $\mu = 0$, 10^{-5}, 10^{-6} and 10^{-7}. Without penalty ($\mu = 0$), the algorithm fails to converge in 100,000 iterations.

Algorithm 2 (PET image recovering). Given the coefficient matrix $\mathbf{E} \in \mathbb{R}_+^{d \times p}$, coincident counts $\mathbf{y} = (y_1, \ldots, y_d) \in \mathbf{Z}_+^d$, and roughness parameter $\mu > 0$, find the intensity vector $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_p) \in \mathbb{R}_+^p$ that maximizes the objective function (3.6).

Scale $\mathbf{E}$ to have unit l_1 column norms.
Compute $|\mathcal{N}_j| = \sum_{k:\{j,k\} \in \mathcal{N}} 1$ and $a_j = -2\mu|\mathcal{N}_j|$ for all $1 \le j \le p$.
Initialize: $\lambda_{0j} \leftarrow 1$, $j = 1, \ldots, p$.
repeat
  $z_{nij} \leftarrow (y_i e_{ij} \lambda_{nj})/(\sum_k e_{ik} \lambda_{nk})$ for all $1 \le i \le d$, $1 \le j \le p$
  for $j = 1$ to p **do**
    $b_{nj} \leftarrow \mu(|\mathcal{N}_j| \lambda_{nj} + \sum_{k \in \mathcal{N}_j} \lambda_{nk}) - 1$
    $c_{nj} \leftarrow \sum_i z_{nij}$
    $\lambda_{n+1,j} \leftarrow (-b_{nj} - \sqrt{b_{nj}^2 - 4a_j c_{nj}})/(2a_j)$
  end for
until convergence occurs

3.3 Multidimensional Scaling

Multidimensional scaling (MDS) was the first statistical application of the MM principle [5, 6]. MDS represents q objects as faithfully as possible in p-dimensional space given a nonnegative weight w_{ij} and a nonnegative dissimilarity measure y_{ij} for each pair of objects i and j. If $\boldsymbol{\theta}^i \in \mathbb{R}^p$ is the position of object i, then the $p \times q$ parameter matrix $\boldsymbol{\theta} = (\boldsymbol{\theta}^1, \ldots, \boldsymbol{\theta}^q)$ is estimated by minimizing the stress function

$$\begin{aligned} f(\boldsymbol{\theta}) &= \sum_{1 \le i < j \le q} w_{ij}(y_{ij} - \|\boldsymbol{\theta}^i - \boldsymbol{\theta}^j\|)^2 \\ &= \sum_{i<j} w_{ij} y_{ij}^2 - 2 \sum_{i<j} w_{ij} y_{ij} \|\boldsymbol{\theta}^i - \boldsymbol{\theta}^j\| \\ &\quad + \sum_{i<j} w_{ij} \|\boldsymbol{\theta}^i - \boldsymbol{\theta}^j\|^2, \end{aligned} \tag{3.8}$$

where $\|\boldsymbol{\theta}^i - \boldsymbol{\theta}^j\|$ is the Euclidean distance between $\boldsymbol{\theta}^i$ and $\boldsymbol{\theta}^j$. The stress function (3.8) is invariant under translations, rotations and reflections of $\mathbb{R}^p$. To avoid translational and rotational ambiguities, we take $\boldsymbol{\theta}^1$ to be the origin and the first $p-1$ coordinates of $\boldsymbol{\theta}^2$ to be 0. Switching the sign of θ_p^2 leaves the stress function invariant. Hence, convergence to one member of a pair of reflected minima immediately determines the other member.

Given these preliminaries, we now review the derivation of the MM algorithm presented in [17]. Because we want to minimize the stress, we majorize it. The middle term in the stress (3.8) is majorized by the Cauchy–Schwarz inequality

$$-\|\boldsymbol{\theta}^i - \boldsymbol{\theta}^j\| \le -\frac{(\boldsymbol{\theta}^i - \boldsymbol{\theta}^j)^t (\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j)}{\|\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j\|}.$$

To separate the parameters in the summands of the third term of the stress, we invoke the convexity of the Euclidean norm $\|\cdot\|$ and the square function s^2. These maneuvers yield

$$\begin{aligned} &\|\boldsymbol{\theta}^i - \boldsymbol{\theta}^j\|^2 \\ &\quad = \left\| \tfrac{1}{2}[2\boldsymbol{\theta}^i - (\boldsymbol{\theta}_n^i + \boldsymbol{\theta}_n^j)] - \tfrac{1}{2}[2\boldsymbol{\theta}^j - (\boldsymbol{\theta}_n^j + \boldsymbol{\theta}_n^j)] \right\|^2 \\ &\quad \le 2\|\boldsymbol{\theta}^i - \tfrac{1}{2}(\boldsymbol{\theta}_n^i + \boldsymbol{\theta}_n^j)\|^2 + 2\|\boldsymbol{\theta}^j - \tfrac{1}{2}(\boldsymbol{\theta}_n^i + \boldsymbol{\theta}_n^j)\|^2. \end{aligned}$$

Assuming that $w_{ij} = w_{ji}$ and $y_{ij} = y_{ji}$, the surrogate function therefore becomes

$$g(\boldsymbol{\theta}|\boldsymbol{\theta}_n) = 2 \sum_{i<j} w_{ij} \left[\left\| \boldsymbol{\theta}^i - \frac{1}{2}(\boldsymbol{\theta}_n^i + \boldsymbol{\theta}_n^j) \right\|^2 - \frac{y_{ij} (\boldsymbol{\theta}^i)^t (\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j)}{\|\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j\|} \right]$$

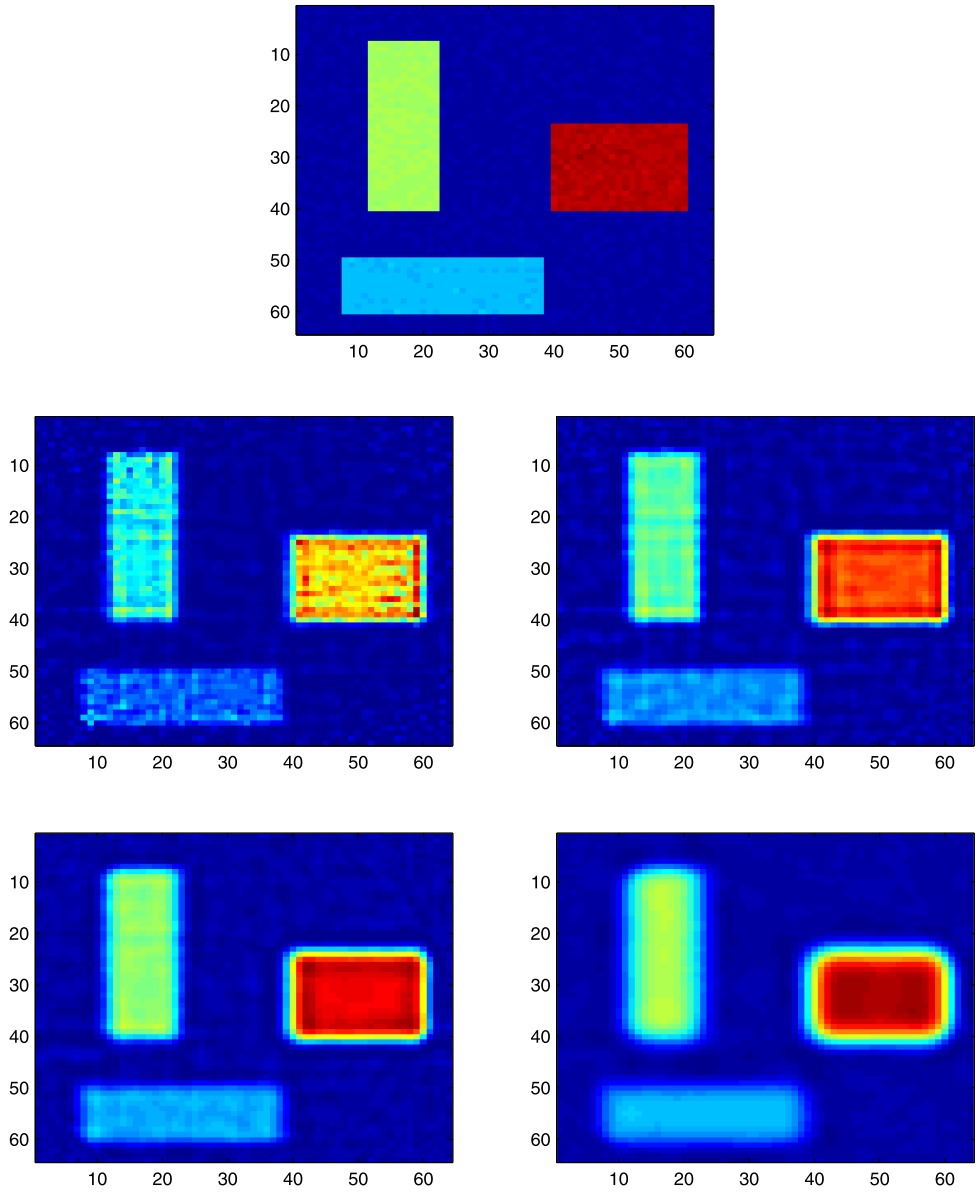

FIGURE 3. *The true PET image (top) and the recovered images with penalties $\mu = 0$ (middle left), 10^{-7} (middle right), 10^{-6} (lower left) and 10^{-5} (lower right).*

$$+2\sum_{i<j} w_{ij}\left[\left\|\boldsymbol{\theta}^j - \frac{1}{2}(\boldsymbol{\theta}_n^i + \boldsymbol{\theta}_n^j)\right\|^2 + \frac{y_{ij}(\boldsymbol{\theta}^j)^t(\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j)}{\|\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j\|}\right]$$

$$= 2\sum_{i=1}^{q}\sum_{j\neq i}\left[w_{ij}\left\|\boldsymbol{\theta}^i - \frac{1}{2}(\boldsymbol{\theta}_n^i + \boldsymbol{\theta}_n^j)\right\|^2 - \frac{w_{ij}y_{ij}(\boldsymbol{\theta}^i)^t(\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j)}{\|\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j\|}\right]$$

up to an irrelevant constant. Setting the gradient of the surrogate equal to the $\mathbf{0}$ vector produces the parallel updates

$$\theta_{n+1,k}^i = \left(\sum_{j\neq i}\left[\frac{w_{ij}y_{ij}(\theta_{nk}^i - \theta_{nk}^j)}{\|\boldsymbol{\theta}_n^i - \boldsymbol{\theta}_n^j\|} + w_{ij}(\theta_{nk}^i + \theta_{nk}^j)\right]\right) \Big/ \left(2\sum_{j\neq i} w_{ij}\right)$$

for all movable parameters θ_k^i.

Algorithm 3 summarizes the parallel organization of the steps. Again the matrix multiplications $\boldsymbol{\Theta}_n^t\boldsymbol{\Theta}_n$ and

Algorithm 3 (MDS). Given weights $\mathbf{W}$ and distances $\mathbf{Y} \in \mathbb{R}^{q \times q}$, find the matrix $\boldsymbol{\Theta} = [\boldsymbol{\theta}^1, \ldots, \boldsymbol{\theta}^q] \in \mathbb{R}^{p \times q}$ which minimizes the stress (3.8).

Precompute: $x_{ij} \leftarrow w_{ij} y_{ij}$ for all $1 \le i, j \le q$
Precompute: $w_{i\cdot} \leftarrow \sum_j w_{ij}$ for all $1 \le i \le q$
Initialize: Draw θ^i_{0k} uniformly on [-1,1] for all $1 \le i \le q$, $1 \le k \le p$
repeat
  Compute $\boldsymbol{\Theta}_n^t \boldsymbol{\Theta}_n$
  $d_{nij} \leftarrow \{\boldsymbol{\Theta}_n^t \boldsymbol{\Theta}_n\}_{ii} + \{\boldsymbol{\Theta}_n^t \boldsymbol{\Theta}_n\}_{jj} - 2\{\boldsymbol{\Theta}_n^t \boldsymbol{\Theta}_n\}_{ij}$ for all $1 \le i, j \le q$
  $z_{nij} \leftarrow x_{ij}/d_{nij}$ for all $1 \le i \ne j \le q$
  $z_{ni\cdot} \leftarrow \sum_j z_{nij}$ for all $1 \le i \le q$
  Compute $\boldsymbol{\Theta}_n(\mathbf{W} - \mathbf{Z}_n)$
  $\theta^i_{n+1,k} \leftarrow [\theta^i_{nk}(w_{i\cdot} + z_{ni\cdot}) + \{\boldsymbol{\Theta}_n(\mathbf{W} - \mathbf{Z}_n)\}_{ik}]/(2w_{i\cdot})$ for all $1 \le i \le p$, $1 \le k \le q$
until convergence occurs

$\boldsymbol{\Theta}_n(\mathbf{W} - \mathbf{Z}_n)$ can be taken care of by the CUBLAS library [26]. The remaining steps of the algorithm are conducted by easily written parallel code.

Table 4 compares the run times in seconds for MDS on the 2005 United States House of Representatives roll call votes. The original data consist of the 671 roll calls made by 401 representatives. We refer readers to the reference [9] for a careful description of the data and how the MDS input 401×401 distance matrix is derived. The weights w_{ij} are taken to be 1. In our notation, the number of objects (House Representatives) is $q = 401$. Even for this relatively small dataset, we see a 27- to 48-fold reduction in total run times, depending on the projection dimension p. Figure 4 displays the results in $p = 3$ dimensional space. The Democratic and Republican members are clearly separated. For $p = 30$, the algorithm fails to converge within 100,000 iterations.

Although the projection of points into $p > 3$ dimensional space may sound artificial, there are situations where this is standard practice. First, MDS is foremost a dimension reduction tool, and it is desirable to keep $p > 3$ to maximize explanatory power. Second, the stress function tends to have multiple local minima in low dimensions [10]. A standard optimization algorithm like MM is only guaranteed to converge to a local minimum of the stress function. As the number of dimensions increases, most of the inferior modes disappear. One can formally demonstrate that the stress has a unique minimum when $p = q - 1$ [4, 10]. In practice, uniqueness can set in well before p reaches $q - 1$. In

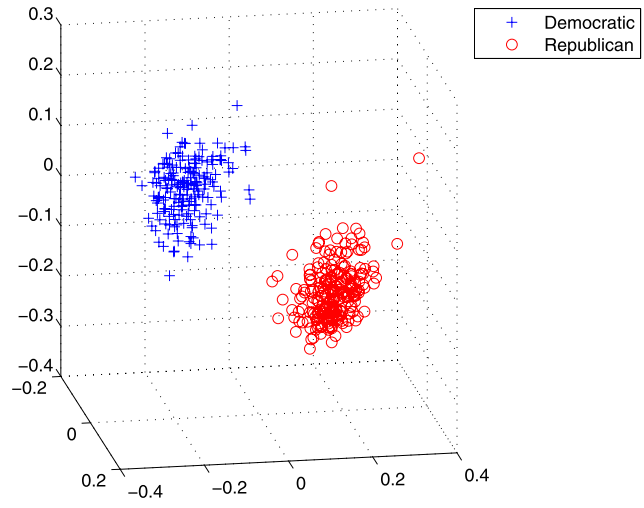

FIGURE 4. *Display of the MDS results with $p = 3$ coordinates on the 2005 House of Representatives roll call data.*

the recent work [41], we propose a "dimension crunching" technique that increases the chance of the MM algorithm converging to the global minimum of the stress function. In dimension crunching, we start optimizing the stress in a Euclidean space $\mathbb{R}^m$ with $m > p$. The last $m - p$ components of each column $\boldsymbol{\theta}^i$ are gradually subjected to stiffer and stiffer penalties. In the limit as the penalty tuning parameter tends to ∞, we recover the global minimum of the stress in $\mathbb{R}^p$. This strategy inevitably incurs a computational burden when m is large, but the MM+GPU combination comes to the rescue.

4. DISCUSSION

The rapid and sustained increases in computing power over the last half century have transformed statistics. Every advance has encouraged statisticians to attack harder and more sophisticated problems. We tend to take the steady march of computational efficiency for granted, but there are limits to a chip's clock speed, power consumption and logical complexity. Parallel processing via GPUs is the technological innovation that will power ambitious statistical computing in the coming decade. Once the limits of parallel processing are reached, we may see quantum computers take off. In the meantime statisticians should learn how to harness GPUs productively.

We have argued by example that high-dimensional optimization is driven by parameter and data separation. It takes both to exploit the parallel capabilities of GPUs. Block relaxation and the MM algorithm often generate ideal parallel algorithms. In our opinion the

TABLE 3
Comparison of run times (in seconds) for a PET imaging problem on the simulated data in [30]

	CPU			GPU				QN(10) on CPU			
Penalty μ	**Iters**	**Time**	**Function**	**Iters**	**Time**	**Function**	**Speedup**	**Iters**	**Time**	**Function**	**Speedup**
0	100,000	14,790	−7337.152765	100,000	282	−7337.153387	52	6549	2094	−7320.100952	n/a
10^{-7}	24,457	3682	−8500.083033	24,457	70	−8508.112249	53	251	83	−8500.077057	44
10^{-6}	6294	919	−15,432.45496	6294	18	−15,432.45586	51	80	29	−15,432.45366	32
10^{-5}	589	86	−55,767.32966	589	2	−55,767.32970	43	19	9	−55,767.32731	10

Notes: The image has $p = 64 \times 64 = 4096$ pixels and is interrogated by $d = 2016$ detectors. The columns labeled Function refer to the converged value of the objective function. The results under the heading QN(10) on CPU invoke quasi-Newton acceleration [42] with 10 secant conditions.

TABLE 4
Comparison of run times (in seconds) for MDS on the 2005 *House of Representatives roll call data*

	CPU			GPU				QN(20) on CPU			
Dim-p	**Iters**	**Time**	**Stress**	**Iters**	**Time**	**Stress**	**Speedup**	**Iters**	**Time**	**Stress**	**Speedup**
2	3452	43	198.5109307	3452	1	198.5109309	43	530	16	198.5815072	3
3	15,912	189	95.55987770	15,912	6	95.55987813	32	1124	38	92.82984196	5
4	15,965	189	56.83482075	15,965	7	56.83482083	27	596	18	56.83478026	11
5	24,604	328	39.41268434	24,604	10	39.41268444	33	546	17	39.41493536	19
10	29,643	441	14.16083986	29,643	13	14.16083992	34	848	35	14.16077368	13
20	67,130	1288	6.464623901	67,130	32	6.464624064	40	810	43	6.464526731	30
30	100,000	2456	4.839570118	100,000	51	4.839570322	48	844	54	4.839140671	n/a

Notes: The number of points (representatives) is $q = 401$. The results under the heading QN(20) on CPU invoke the quasi-Newton acceleration [42] with 20 secant conditions.

MM algorithm is the more versatile of the two generic strategies. Unfortunately, block relaxation does not accommodate constraints well and may generate sequential rather than parallel updates. Even when its updates are parallel, they may not be data separated. The EM algorithm is one of the most versatile tools in the statistician's toolbox. The MM principle generalizes the EM algorithm and shares its positive features. Scoring and Newton's methods become impractical in high dimensions. Despite these arguments in favor of MM algorithms, one should always keep in mind hybrid algorithms such as the one we implemented for NNMF.

Although none of our datasets is really large by today's standards, they do demonstrate that a good GPU implementation can easily achieve one to two orders of magnitude improvement over a single CPU core. Admittedly, modern CPUs come with two to eight cores, and distributed computing over CPU-based clusters remains an option. But this alternative also carries a hefty price tag. The NVIDIA GTX280 GPU on which our examples were run drives 240 cores at a cost of several hundred dollars. High-end computers with eight or more CPU nodes cost thousands of dollars. It would take 30 CPUs with eight cores each to equal a single GPU at the same clock rate. Hence, GPU cards strike an effective and cost-efficient balance.

In the three test examples, we report performance results on the GTX 280 GPU card in single precision. While this two-year-old card is the first to support double-precision computation, its performance is suboptimal. As the numerical results show, single-precision speedups are gained at a potential loss of accuracy and, in rare cases, single precision may lead to an inferior mode, for example, the $\mu = 10^{-7}$ case in Table 3. In some preliminary experimentation on the same PET imaging algorithm, we find that a double-precision implementation on GTX 280 runs at about $1/3$ the speed of single precision. However, GPU technology is advancing rapidly. This year's GTX 480 GPU card drives twice as many cores as the GTX 280 and offers much improved double-precision support. On the same desktop system as in Table 1, a GTX 480 delivers 89-fold speedup in single-precision and 43-fold speedup in double-precision over the CPU code. Very recently released, the Tesla C2050 GPU sports a peak double-precision floating point performance (515 Gflops) that is three times that of the GTX 480 (168 Gflops). The challenge for the statistics community is to tackle more complicated statistical models on bigger datasets. Computational statisticians will have to judge the speed versus precision trade-off problem by problem.

The simplicity of MM algorithms often comes at a price of slow (at best linear) convergence. Our MDS, NNMF and PET (without penalty) examples are cases in point. Slow convergence is a concern as statisticians head into an era dominated by large datasets and high-dimensional models. Think about the scale of the Netflix data matrix. The speed of any iterative algorithm is determined by both the computational cost per iteration and the number of iterations until convergence. GPU implementation reduces the first cost. Computational statisticians also have a bag of software tricks to decrease the number of iterations [11, 12, 15, 21, 23, 24, 38]. For instance, the recent article [42] proposes a quasi-Newton acceleration scheme particularly suitable for high-dimensional problems. The scheme is off-the-shelf and broadly applies to any search algorithm defined by a smooth algorithm map. The acceleration requires only modest increments in storage and computation per iteration. Tables 3 and 4 also list the results of this quasi-Newton acceleration of the CPU implementation for the MDS and PET examples. As the tables make evident, quasi-Newton acceleration significantly reduces the number of iterations until convergence. The accelerated algorithm always locates a better mode while cutting run times compared to the unaccelerated algorithm. We have tried the quasi-Newton acceleration on our GPU hardware with mixed results. We suspect that the lack of full double precision on the GPU is the culprit. When full double precision becomes widely available, the combination of GPU hardware acceleration and algorithmic software acceleration will be extremely potent.

Successful acceleration methods will also facilitate attacking another nagging problem in computational statistics, namely multimodality. No one knows how often statistical inference is fatally flawed because a standard optimization algorithm converges to an inferior mode. The current remedy of choice is to start a search algorithm from multiple random points. Algorithm acceleration is welcome because the number of starting points can be enlarged without an increase in computing time. As an alternative to multiple starting points, our recent article [41] suggests modifications of several standard MM algorithms that increase the chance of locating better modes. These simple modifications all involve variations on deterministic annealing [37].

Our treatment of simple classical examples should not hide the wide applicability of the powerful MM+

GPU combination. A few other candidate applications include penalized estimation of haplotype frequencies in genetics [1], construction of biological and social networks under a random multigraph model [29], and data mining with a variety of models related to the multinomial distribution [43]. Many mixture models will benefit as well from parallelization, particularly in assigning group memberships. Finally, parallelization is hardly limited to optimization. We can expect to see many more GPU applications in MCMC sampling. Given the computationally intensive nature of MCMC, the ultimate payoff may even be higher in the Bayesian setting than in the frequentist setting. For example, in a recent study [35], GPU implementations deliver up to a 140-fold speedup in Bayesian fitting of massive mixture models. Of course, realistically these future triumphs will require a great deal of thought, effort, and education. There is usually a desert to wander and a river to cross before one reaches the promised land.

ACKNOWLEDGMENTS

The authors thank the editor and three reviewers for their valuable suggestions for improving the manuscript. M. S. acknowledges support from NIH Grant R01 GM086887. K. L. was supported by United States Public Health Service Grants GM53275 and MH59490.

REFERENCES

[1] AYERS, K. L. and LANGE, K. L (2008). Penalized estimation of haplotype frequencies. *Bioinformatics* **24** 1596–1602.

[2] BERRY, M. W., BROWNE, M., LANGVILLE, A. N., PAUCA, V. P. and PLEMMONS, R. J. (2007). Algorithms and applications for approximate nonnegative matrix factorization. *Comput. Statist. Data Anal.* **52** 155–173. MR2409971

[3] BUCKNER, J., WILSON, J., SELIGMAN, M., ATHEY, B., WATSON, S. and MENG, F. (2010). The gputools package enables GPU computing in R. *Bioinformatics* **26** 134–135.

[4] DE LEEUW, J. (1992). Fitting distances by least squares. Unpublished manuscript.

[5] DE LEEUW, J. and HEISER, W. J. (1977). Convergence of correction matrix algorithms for multidimensional scaling. In *Geometric Representations of Relational Data* 133–145. Mathesis Press, Ann Arbor, MI.

[6] DE LEEUW, J. (1977). Applications of convex analysis to multidimensional scaling. In *Recent Developments in Statistics (Proc. European Meeting Statisticians, Grenoble, 1976)* 133–145. North-Holland, Amsterdam. MR0478483

[7] DE LEEUW, J. (1994). Block relaxation algorithms in statistics. In *Information Systems and Data Analysis*. Springer, Berlin.

[8] DEMPSTER, A. P., LAIRD, N. M. and RUBIN, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm (with discussion). *J. Roy. Statist. Soc. Ser. B* **39** 1–38. MR0501537

[9] DIACONIS, P., GOEL, S. and HOLMES, S. (2008). Horseshoes in multidimensional scaling and local kernel methods. *Ann. Appl. Statist.* **2** 777–807. MR2516794

[10] GROENEN, P. J. F. and HEISER, W. J. (1996). The tunneling method for global optimization in multidimensional scaling. *Pshychometrika* **61** 529–550.

[11] JAMSHIDIAN, M. and JENNRICH, R. I. (1993). Conjugate gradient acceleration of the EM algorithm. *J. Amer. Statist. Assoc.* **88** 221–228. MR1212487

[12] JAMSHIDIAN, M. and JENNRICH, R. I. (1997). Acceleration of the EM algorithm by using quasi-Newton methods. *J. Roy. Statist. Soc. Ser. B* **59** 569–587. MR1452026

[13] KOREN, Y., BELL, R. and VOLINSKY, C. (2009). Matrix factorization techniques for recommender systems. *Computer* **42** 30–37.

[14] LANGE, K. L. and CARSON, R. (1984). EM reconstruction algorithms for emission and transmission tomography. *J. Comput. Assist. Tomogr.* **8** 306–316.

[15] LANGE, K. L. (1995). A quasi-Newton acceleration of the EM algorithm. *Statist. Sinica* **5** 1–18. MR1329286

[16] LANGE, K. L. (2004). *Optimization*. Springer, New York. MR2072899

[17] LANGE, K. L., HUNTER, D. R. and YANG, I. (2000). Optimization transfer using surrogate objective functions (with discussion). *J. Comput. Graph. Statist.* **9** 1–59. MR1819865

[18] LEE, A., YAN, C., GILES, M. B., DOUCET, A. and HOLMES, C. C. (2009). On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. Technical report, Dept. Statistics, Oxford Univ.

[19] LEE, D. D. and SEUNG, H. S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature* **401** 788–791.

[20] LEE, D. D. and SEUNG, H. S. (2001). Algorithms for nonnegative matrix factorization. In *NIPS* 556–562. MIT Press, Cambridge, MA.

[21] LIU, C. and RUBIN, D. B. (1994). The ECME algorithm: A simple extension of EM and ECM with faster monotone convergence. *Biometrika* **81** 633–648. MR1326414

[22] MCLACHLAN, G. J. and KRISHNAN, T. (2008). *The EM Algorithm and Extensions*, 2nd ed. Wiley, Hoboken, NJ. MR2392878

[23] MENG, X. L. and RUBIN, D. B. (1993). Maximum likelihood estimation via the ECM algorithm: A general framework. *Biometrika* **80** 267–278. MR1243503

[24] MENG, X. L. and VAN DYK, D. (1997). The EM algorithm—an old folk-song sung to a fast new tune (with discussion). *J. Roy. Statist. Soc. Ser. B* **59** 511–567. MR1452025

[25] MIT center for biological and computational learning. CBCL Face Database #1. Available at http://cbcl.mit.edu/.

[26] NVIDIA (2008). NVIDIA CUBLAS Library.

[27] NVIDIA (2008). NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 2.0.

[28] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E. and PURCELL, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* **26** 80–113.

[29] RANOLA, J. M., AHN, S., SEHL, M. E., SMITH, D. J. and LANGE, K. L. (2010). A Poisson model for random multigraphs. *Bioinformatics* **26** 2004–2011.

[30] ROLAND, C., VARADHAN, R. and FRANGAKIS, C. E. (2007). Squared polynomial extrapolation methods with cycling: An application to the positron emission tomography problem. *Numer. Algorithms* **44** 159–172. MR2334694

[31] SILBERSTEIN, M., SCHUSTER, A., GEIGER, D., PATNEY, A. and OWENS, J. D. (2008). Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd Annual International Conference on Supercomputing* 309–318. ACM, New York.

[32] SINNOTT-ARMSTRONG, N. A., GREENE, C. S., CANCARE, F. and MOORE, J. H. (2009). Accelerating epistasis analysis in human genetics with consumer graphics hardware. *BMC Res. Notes* **2** 149.

[33] SUCHARD, M. A. and RAMBAUT, A. (2009). Many-core algorithms for statistical phylogenetics. *Bioinformatics* **25** 1370–1376.

[34] SUCHARD, M. A., HOLMES, C. and WEST, M. (2010). Some of the what?, why?, how?, who and where? of graphics processing unit computing for Bayesian analysis. *ISBA Bull.* **17** 12–16.

[35] SUCHARD, M. A, WANG, Q., CHAN, C., FRELINGER, A. and West, M. (2010). Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *J. Comput. Graph. Statist.* **19** 418–438.

[36] TIBBITS, M. M., HARAN, M. and LIECHTY, J. C. (2010). Parallel multivariate slice sampling. *Statist. Comput.* To appear.

[37] UEDA, N. and NAKANO, R. (1998). Deterministic annealing EM algorithm. *Neural Networks* **11** 271–282.

[38] VARADHAN, R. and ROLAND, C. (2008). Simple and globally convergent methods for accelerating the convergence of any EM algorithm. *Scand. J. Statist.* **35** 335–353. MR2418745

[39] VARDI, Y., SHEPP, L. A. and KAUFMAN, L. (1985). A statistical model for positron emission tomography (with discussion). *J. Amer. Statist. Assoc.* **80** 8–37. MR0786595

[40] WU, T. T. and LANGE, K. L. (2010). The MM alternative to EM. *Statist. Sci.* To appear.

[41] ZHOU, H. and LANGE K. L. (2009). On the bumpy road to the dominant mode. *Scand. J. Statist.* DOI: 10.1111/j.1467-9469.2009.00681.x.

[42] ZHOU, H., ALEXANDER, D. and LANGE, K. L. (2009). A quasi-Newton acceleration for high-dimensional optimization algorithms. *Statist. Comput.* DOI: 10.1007/s11222-009-9166-3.

[43] ZHOU, H. and LANGE, K. L. (2010). MM algorithms for some discrete multivariate distributions. *J. Comput. Graph. Statist.* **19** 645–665.